

EXPRESSJS

Express

Overview

What is [Express](#)? Well, **Express is just a library/framework built for Node.js that allows us to more easily handle requests and build a robust server.**

There are tons of ways to build apps with Express! Unlike other MVC frameworks (like Codeigniter and Rails), Express is quite flexible in the way it can be configured. This provides tons of customization for experienced developers.

However, for someone more on the beginner side, Express can seem like an endless maze of Javascript. In this section, we will show you the easiest way to configure Express right out of the box so we can start building awesome apps. Keep in mind that there are plenty of other ways to configure Express but the fundamentals presented here are more than enough to grasp the core concepts!

We are going to build Express applications from scratch, so you will need to have a working knowledge of the following before starting this chapter:

- Client-Server Model
- The Traditional HTTP request/response cycle
- What is Node.js?
 - What does "require" do?
 - What is module.exports?
 - What are Node modules (middleware)?

Express

What is Express?

Express.js is a framework built in Javascript using Node.js as the server component. It is also the E in the **MEAN** stack, so it must be important! Unlike most other MVC frameworks (like Codeigniter and Rails), **Express isn't exclusively MVC**. It's actually more so just a set of tools that allows us to create a more robust Node Server. There are a few different ways to build an MVC framework with Express, and **we will eventually show you how we do it**. However, we believe the best way to learn **Express** is to start with the basics before working our way up.

Your first Express project:

Create a folder called "Hello_Express" and give it a file called "server.js". This file will be our Node server file.

Since Express is a node module, run `npm install express` while in the Hello_Express directory.

Note: If you are on a Mac you may need to run this command as sudo.

All that this did was go out and install the express module (as well as all of the modules that express is dependent on) in your project folder. Your project folder should now contain a big new scary "node_modules" folder in addition to your server.js file! Feel free to poke around and see what is inside, but do not delete or change anything. Try to trace where `require("express")` from our server.js code goes and what it comes back with.

Hello_Express/server.js

```
// Load the express module and store it in the variable express (Where do you think this comes from?)
var express = require("express");
console.log("Let's find out what express is", express);
// invoke express and store the result in the variable app
var app = express();
console.log("Let's find out what app is", app);
// use app's get method and pass it the base route '/' and a callback
app.get('/', function(request, response) {
  // just for fun, take a look at the request and response objects
  console.log("The request object", request);
  console.log("The response object", response);
  // use the response object's .send() method to respond with an h1
  response.send("<h1>Hello Express</h1>");
})
// tell the express app to listen on port 8000, always put this at the end of your server.js file
app.listen(8000, function() {
  console.log("listening on port 8000");
})
```

Run the server.js file using node (but use nodemon, it will make your life easier) and see the magic happen!

```
nodemon server.js
```

Views

What does the client see?

HTML/CSS/JS! Remember that the client only sees HTML/CSS/JS? Generally, there are two ways to serve HTML/CSS/JS - through Static Content or Templates.

Static Content -- Serving a static HTML/CSS/JS file from the backend in response to a request.


```
    {name: "Jay", email: "jay@codingdojo.com"},
    {name: "Brendan", email: "brendan@codingdojo.com"},
    {name: "Andrew", email: "andrew@codingdojo.com"}
  ];
  response.render('users', {users: users_array});
})
```

Notice we are passing a JavaScript object to the `response.render()` method. That way, when we pass a piece of data to a view, **every key-value pair within the larger piece of data becomes its own variable**.

Let's create a new folder called "views" in our project directory. This folder is where we are going to put all of our view files.

Next let's make a view called `users.ejs` and put in it the following:

```
<html>
<body>
  <h2>Here are all the users:</h2>
  <% for (var x in users) { %>
    <h3>Name: <%= users[x].name %></h3>
    <h4>Email: <%= users[x].email %></h4>
    <hr>
  <% } %>
</body>
</html>
```

The `<% %>` tags are the delimiter for the embedded JavaScript. Using these tags allows us to run JavaScript code that can be **embedded into the HTML** document we are making. **Notice the `<% %>` tags allow us to enter JavaScript code, and the `<%= %>` tags actually print the JavaScript code to the document.** This is a key difference. **You'll use the tags with the equal sign (=) to actually print values, whereas you'll use the tags without the equals sign to invoke loops or use logic (anything that involves JavaScript but doesn't output code).** Embedded JavaScript should be pretty quick to get up and running out of the box. Play around with it and move on when you're ready!!

Express Cars and Cats

You now have the tools necessary to repeat the Cars and Cats assignment from Node, but with Express!

Take the time to appreciate what a difference a framework makes.

For this assignment, you will need a static directory. You will not need routes, ejs, nor a views directory.

Create four html documents in your static directory. These files will be served with the following urls. Why? Because we're requesting static content, and because of our Express static middleware, our server knows to find static files in the static directory.

`localhost:8000/cars.html` - A simple HTML page that shows some cool pictures of different cars. These car pictures should be stored in your static directory. DON'T just link to pictures of cars stored somewhere else! Even better, put them in a directory called 'images' inside of your static directory.

`localhost:8000/cats.html` - A simple HTML page with some cool pictures of cats. Again, make sure these pictures are stored on your server.

`localhost:8000/form.html` - A simple form where the user can add new car information. For this page, there is no need to have the form do anything. Simply display the form there.

Also, add a basic html file in your static directory called index.html. What happens when you navigate to the root route `localhost:8000`?

EJS Cars and Cats

Repeat the previous assignment, but this time use EJS and include a views directory.

By telling Express where to find your views directory and that you are using EJS, you will be able to write the following routes in your server:

Have `/cars` show your pictures of cars.

Have `/cats` show your pictures of cats.

Have `/cars/new` show a form to create a new car. The form does not have to do anything yet.

Keep your index.html file in the static directory. It should render even when your server does not explicitly handle the `/` route

EJS Cat Data




In this assignment, we are going to focus on displaying data about our particular cats.

As before, when we navigate to `localhost:8000/cats`, we should see pictures of all our cats. This time, when we click the pictures, they will navigate us to a unique route in the server. **This route may be hardcoded for now.**

Page 1

localhost:8000/cats

EJS cats




Have all the cats shown on the /cats route. Clicking the cat will take the user to a unique route.

Page 1

localhost:8000/cuddles

Details Page for Cuddles



For each cat's route, render the template details.ejs. Populate it with data about the particular cat. You may have this data hardcoded in the server file.

Favorite food: Spaghetti

Age: 3

Sleeping spots:

under the bed

in a sunbeam

Include at least one array in the cat's data and iterate over it in the template using ejs.


```
app.use(bodyParser.urlencoded({extended: true}));
```

Let's say in an `index.ejs` view file we had a form that looked like this:

```
<form action='/users' method='post'>
  Name: <input type='text' name='name'>
  Email: <input type='text' name='email'>
  <input type='submit' value='create user'>
</form>
```

We're in the **MEAN** stack, right? JavaScript runs the universe now. What do we expect our form data to be? You guessed it: **JSON**. Here's how we get form data:

```
// route to process new user form data:
app.post('/users', function (req, res){
  console.log("POST DATA \n\n", req.body)
  //code to add user to db goes here!
  // redirect the user back to the root route.
  res.redirect('/')
});
```

Now we will go to the **terminal window our server is running on** and check it out...

There you have it! **req.body** is a **JSON object that contains the data from our form**. Not so bad.

Data from URL (GET data)

Let's say we wanted to show the information of a specific user. The **RESTful route** for this would be:

```
users/:id //where :id is the id of a particular user. HTTP method is GET
```

Setting this up in Express is easy; in our `server.js` file we would just **add the route**:

```
app.get("/users/:id", function (req, res){
  console.log("The user id requested is:", req.params.id);
  // just to illustrate that req.params is usable here:
  res.send("You requested the user with id: " + req.params.id);
  // code to get user from db goes here, etc...
});
```

Note: This illustrates **accessing data from the URL**. If you want to test this out, **add a button on the view that sends a request to `/users/1`** (trying to access the page with information about the user with `id = 1`). Note that a post request to `/users` and a get request to `/users` are two completely different routes because a route is made up of the **verb + the URL**.

Any data you wish to pass via the URL must be indicated by a `'.'`. It will then be available in the **req.params object**.

Session data

Session data should be used as little as possible (depending on the situation); using the proper request and response cycle, you will find that there is rarely a need for session data. Using session where unnecessary incurs needless overhead and is heavily discouraged. We teach it briefly here to give you some exposure to it, but it is heavily

recommended against at this stage. If you do need to use it, you have to install and require the *express-session* module first. Common use cases include 'logging in' a user, and storing their 'user_id' into session to be able to retrieve them from different routes.

After installing go to your server.js file and require it like so:

```
// new code:
var session = require('express-session');
// original code:
var app = express();
// more new code:
app.use(session({
  secret: 'keyboardkitteh',
  resave: false,
  saveUninitialized: true,
  cookie: { maxAge: 60000 }
}))
```

Now, within any of your routes, there will be an object called **req.session**. It is an object and you can assign properties to it like normal, in this example, we are storing into req.session.name the value of the post data req.body.name:

```
app.post('/users', function (req, res){
  // set the name property of session.
  req.session.name = req.body.name;
  console.log(req.session.name);
  //code to add user to db goes here!
  // redirect the user back to the root route.
  res.redirect('/');
});
```

And now **req.session.name** will be available to any other route afterward (until our session expires, more on this later).

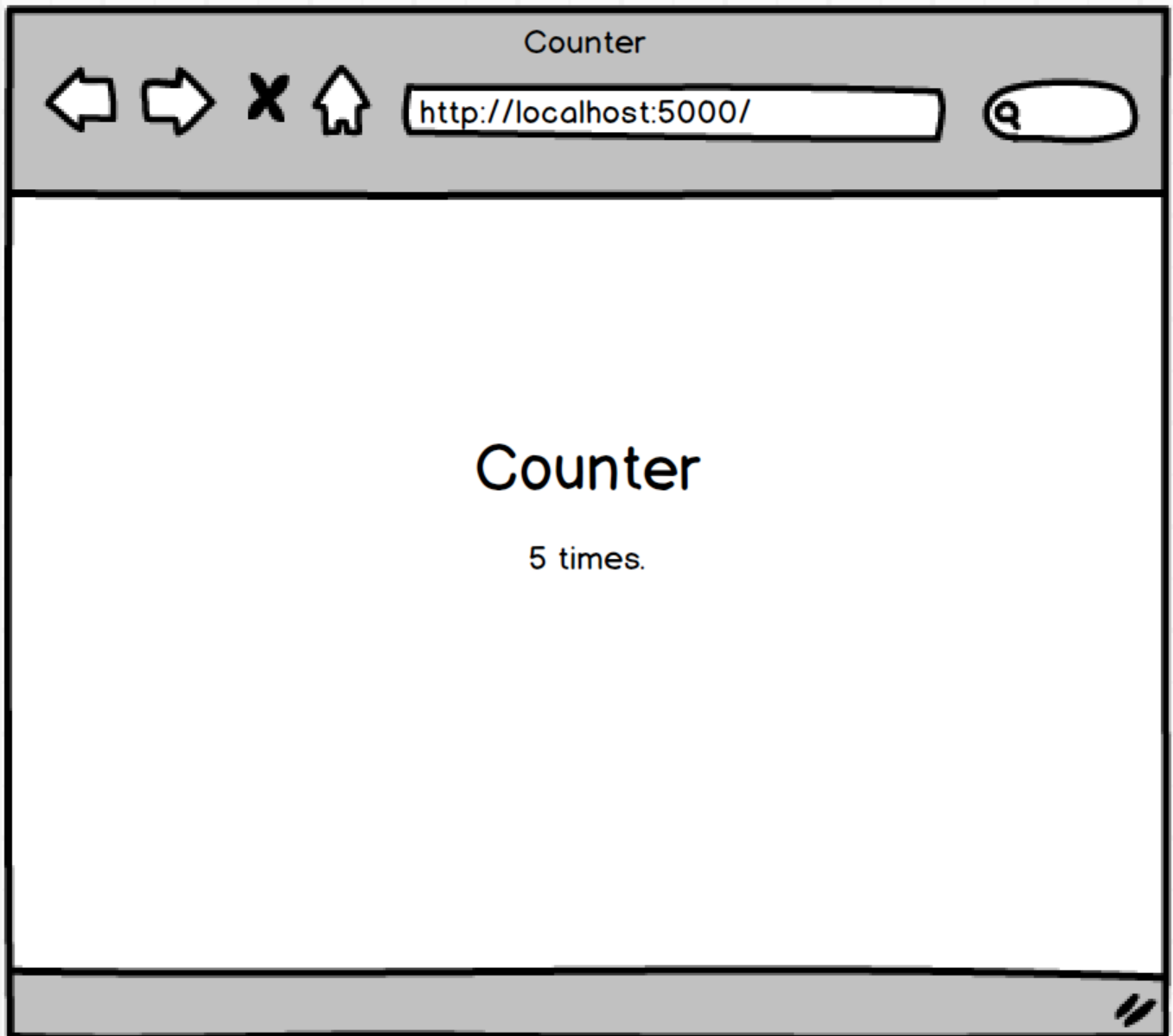
package.json, bower.json

Package managers, such as NPM and Bower, keep track of what modules you are using through JSON files. In general, these files contain information about the project overall, as well as which modules have been downloaded for the project, specifically. If you choose to move a project from one machine to another e.g. into GitHub (or submitting to the site), these .json files minimize the need for copying all of the dependent modules that we download from the package management sites, keeping the overall size of the project that we pass much smaller.

We have dealt with three packages that we include in a lot of our files so far that aren't pre-installed with node: express, ejs, and body-parser. Let's use NPM's package.json file!

Create an empty folder called TestProject

- navigate to that folder using your terminal/command-prompt/bash.
- in terminal/command-prompt/bash type `npm init -y`



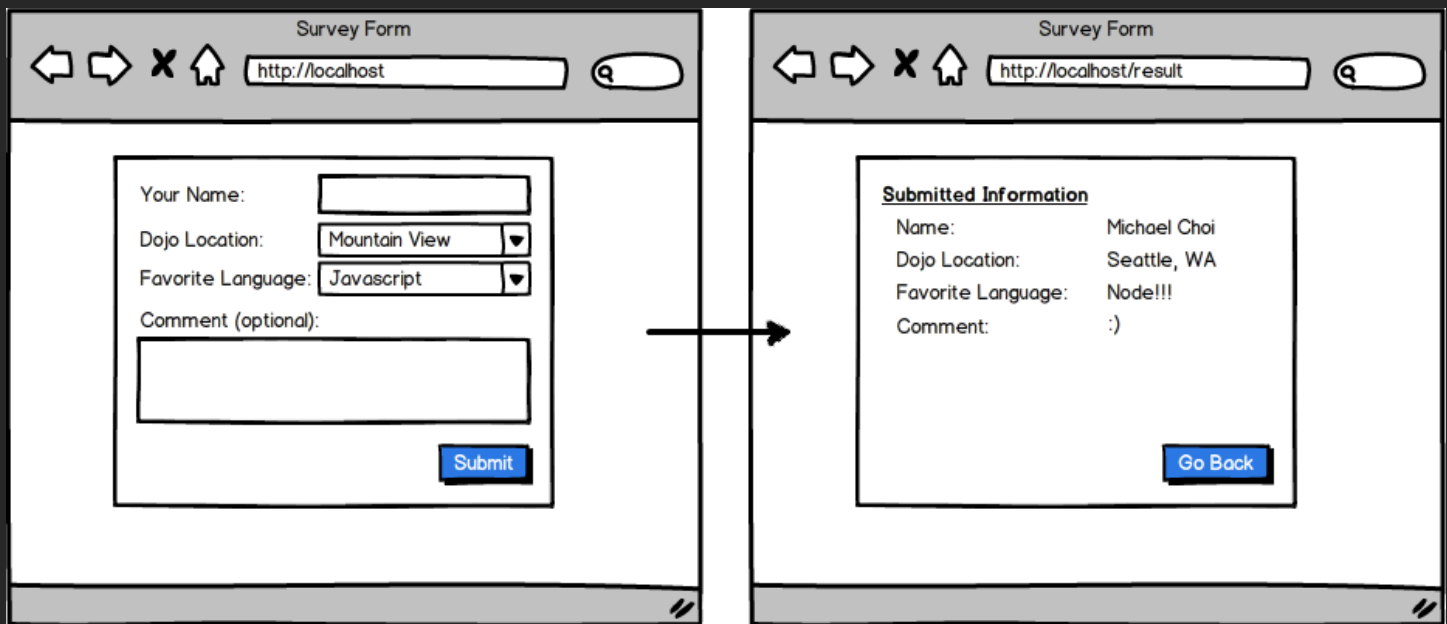
Ninja Level 1

Add a +2 button underneath the counter that reloads the page and increments counter by 2. Add another route to handle this functionality.

Ninja Level 2

Add a reset button that resets the counter back to 1. Add another route to handle this functionality.

Assignment: Survey Form



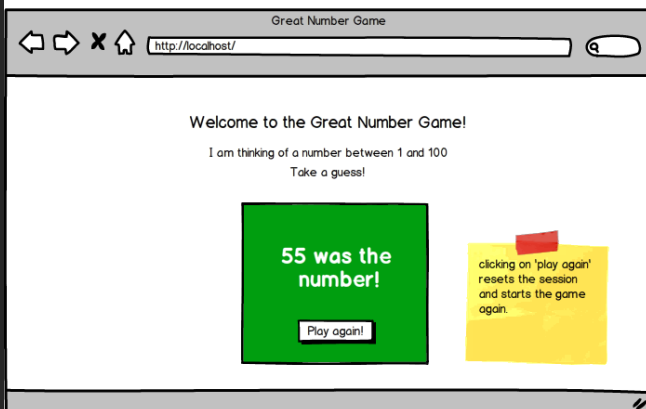
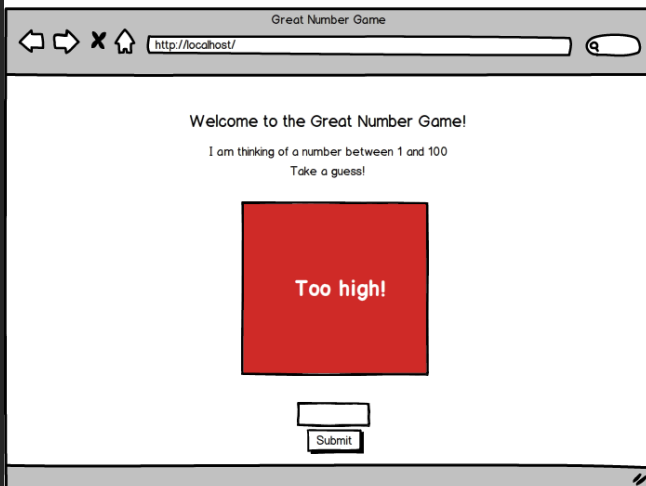
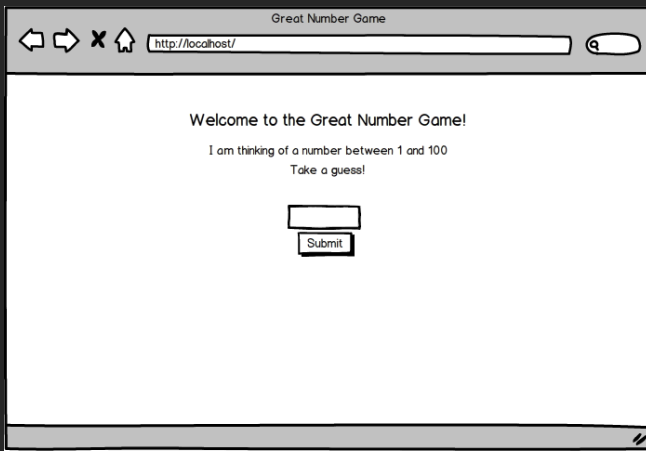
Before you start coding this, first outline or write down the steps of accomplishing this.

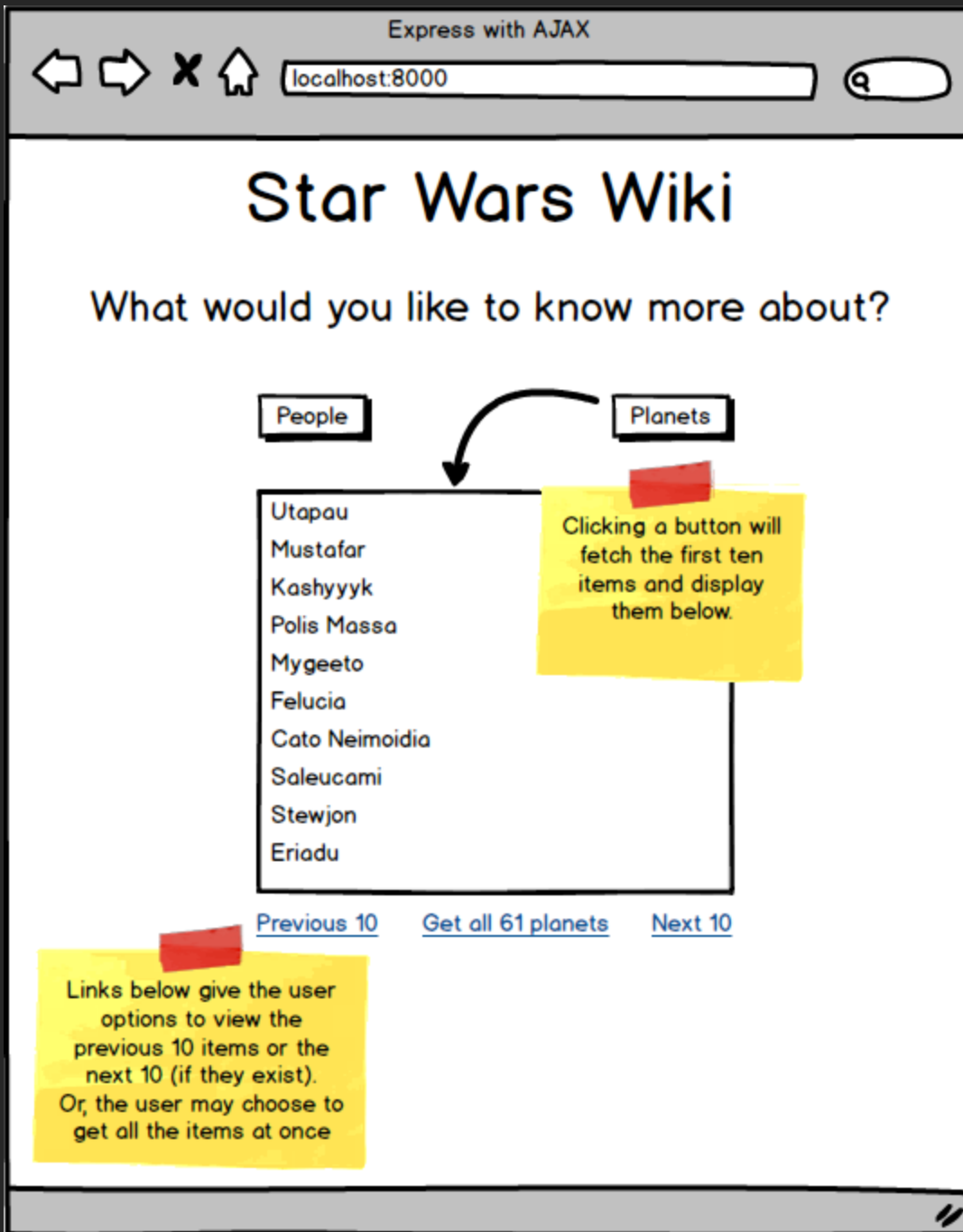
For example

1. Have the server render `views/index.ejs` that has the form for the user to fill out
2. The user fills out the form and submits
3. The submitted form gets sent to `/result`
4. The server recognizes when someone posts things to `/result`, grabs information from the POST, and sends the POST data back as it renders `views/results.ejs`

Assignment: Great Number Game

Create a site that when a user loads it creates a random number between 1-100 and stores the number in **session**. Allow the user to guess at the number and tell them when they are too high or too low. If they guess the correct number tell them and offer to play again.





Previously, we used jQuery to make requests to an API. Therefore, we may decide to have our server send `index.ejs` to the client, and we could have `index.ejs` import jQuery and make requests to the API when the button is clicked. The code below shows one way this could be done:

`index.ejs`

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script>
  $(document).ready(function() {
    $('#peopleBtn').click(function() {
      $.get('http://swapi.co/api/people', function(data) {
        // log the data to be sure we have it before we dive into manipulating the DOM
        console.log("got the data", data);
      });
    });
  });
</script>
```

```

        }, 'json');
    });
});
</script>
<body>
    <button id="peopleBtn">People</button>
</body>

```

However, what happens when we try this? Sadly, we'll see this message in our console:

```

✖ Failed to load http://swapi.co/api/people: Redirect from 'http://swapi.co/api/people' to 'https://swapi.co/a swapi.html:1
pi/people' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'null' is therefore not allowed access.

```

Cross-Origin Resource Sharing (CORS) and Access Control Origin

The access-control-origin error is caused by a configuration setting on certain API servers, where the API provider has configured the server to **control** incoming requests. It allows only those from certain origin types to access the API. This means that our client-side JavaScript code cannot directly contact the API server with requests.

To bypass this, we have to make an AJAX request from our client-side page *to our own server*. Notice that in the code snippet below, the button click is triggering a request to the route `/people`, which is a route we make on our own server. Let's have our **server** make the request to the API and *pass back the response*.

index.ejs

```

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script>
    $(document).ready(function() {
        $('#peopleBtn').click(function() {
            // let's make the request to our OWN server!
            $.get('/people', function(data) {
                // log the data to be sure we have it before we dive into manipulating the DOM
                console.log("got the data", data);
            }, 'json');
        });
    });
</script>
<body>
    <button id="peopleBtn">People</button>
</body>

```

To make requests from our server to different servers, we can use the **Axios** package. It is not the only option available to us, but this one is nice because it uses **promises**.

To use a promise, we'll need two **callbacks**: one if the request is successful, and another if it fails. We place these callbacks in the `.then()` and `.catch()` methods respectively, which are chained after the request.

Visit the optional [JavaScript Advanced chapter](#) to learn more about promises if you haven't done so already.

To get started, use npm to install Axios, and then require it into your server.

```
npm install axios
```

server.js

```
... other server code
const axios = require('axios');
app.get('/people', function(req, res){
  // use the axios .get() method - provide a url and chain the .then() and .catch() methods
  axios.get(url)
    .then(data => {
      // log the data before moving on!
      console.log(data);
      // rather than rendering, just send back the json data!
      res.json(data);
    })
    .catch(error => {
      // log the error before moving on!
      console.log(error);
      res.json(error);
    })
});
```

Arrow functions

In the code snippet above, we are using ES6 **arrow functions**. They are a more concise way to write anonymous functions! Remember that anonymous functions do not have names, but they may be stored in a variable. Depending on the code within them, parentheses, curly brackets, and the `return` statement are not always necessary. However, it does not hurt if you choose to include them.

```
// es5 style
var anonES5 = function(parameter){
  return parameter + 5;
}
// arrow functions
const anonES6 = parameter => parameter + 5;
// curly brackets are not required if there is only one expression
// parentheses are not required if there is only one parameter
// the return is implicit with just one line
const twoParams = (parameter1, parameter2) => {
  parameter1 += 5;
  return parameter1 + parameter2;
}
// with more parameters, parentheses are required
// with more lines of code, curly brackets are required
```

We will be using arrow functions extensively when we work with Angular because they do not bind their own `this`, so the meaning of `this` is more intuitive. Arrow functions use what is called the `lexical this`, which means `this` will always refer to the `this` of the code that contains the arrow function.

Star Wars API

Objectives

1. Gain familiarity with CORS and access control origin
 2. Make requests to an API from the server
 3. Make recursive API calls
-

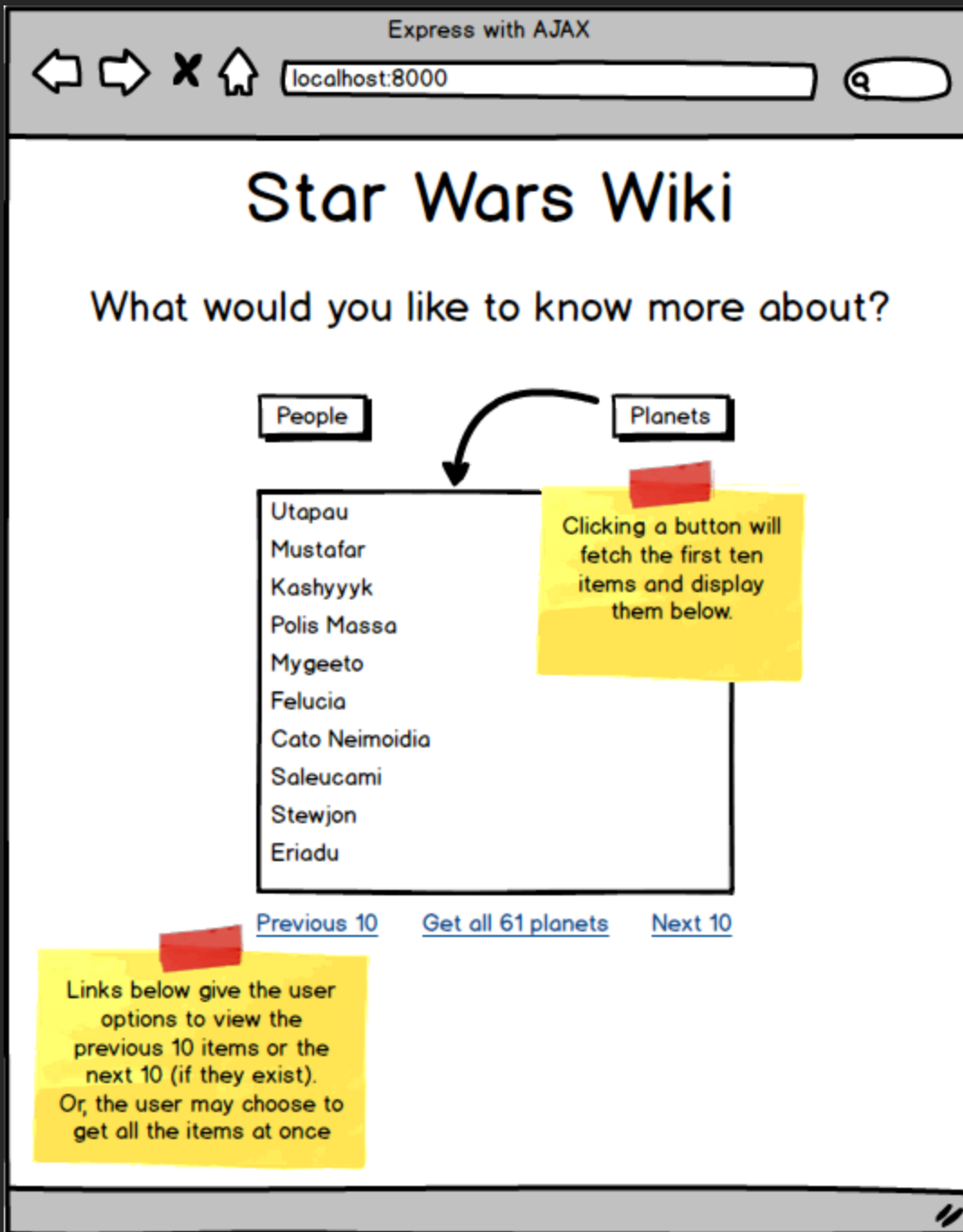
Use the [Star Wars API](#) to gather information about the people and planets of Star Wars based on what the user requests. Upon first visiting the web page, the user should see two buttons - one to receive information about people, one to receive information about planets.

Remember that we should not refresh the entire page. Instead of rendering a new page, have your server respond with JSON by using `res.json()`. The client should use the JSON data to manipulate the DOM, which means only a portion of the page needs to be updated.

You will notice that the API gives data in chunks. For example, making a request to <https://swapi.co/api/people> will respond with an object with an array that contains only ten people. The `count` attribute, however, tells us that there are 87 people we could access. We need to follow the url provided in the `next` attribute to get the next ten.

This assignment asks you to provide links that allow the user to fetch the next ten or previous ten results. Additionally, provide a link that fetches all the people or all the planets! This will require recursion. Notice that this will take a while to load. The intention is to give you experience with recursion and response times so that you may make wise choices about user experience in the future.

```
{
  "count": 87,
  "next": "https://swapi.co/api/people/?page=2",
  "previous": null,
  "results": [
    {
      "name": "Luke Skywalker",
      "height": "172",
      ....
    } ..... // and 9 more people objects
  ],
}
```



BONUS: Rather than providing links to get the next or previous ten items, place a scroll bar to look at the list of fetched items. When the scroll bar hits the bottom of the list, make an API call to fetch the next ten and append them to the list.